CS2030S PE1

AY24/25 sem 2

github.com/mendax1234

Intro to OOP

1. **(Constructor)**:
   - If your class includes a **constructor with parameters**, you are required to **provide arguments** when creating an object using that constructor.
   - In the Constructor of a class, always think about what are the **necessary fields that should be included**.
2. **(Initialization)**: Any *reference* variable that is not initialized will have *null*. Any *primitive* type variable will have either 0 or false (boolean).
3. **(Java)**: **Java** is a **statically typed** and **strongly typed** language.
   - **Statically typed**: the variable can only hold values of the **declared type**. (Any subtype of the declared type is allowed).
   - **Strongly typed**: If there is any problem with the program, it is **not due to the type**. e.g., **no implicit narrowing conversion** is allowed.
   - Java is a **strongly typed language**, but it allows **widening type conversion** and will do this **automatically without explicit casting**.
   - In Java, two types without a subtype relationship **cannot** be casted.
   - **For each loop**: `for (type variableName : arrayName)` **array must be a Java Array**, the CS2030S own `Seq` doesn't support this *for-each* loop.
   - **Nested method calling**: In Java, the nested method call is executed from **left to right**. e.g. `Box.of("string").map(new StringLength()).map(new AddOne());`, the left `.map` will be executed first.
   - **Min Max function**: In Java, we have `min/max = Math.min/max(Number, Number)`
   - **method return**: Suppose the return type of a method is **T**, inside this method, you can actually return the **subtype** of **T**.
4. **Information Hiding**:
   - **fields** should be declared as `private`
   - **methods** should be declared as `public`

More on OOP

1. **Modifier**
   - **Access Modifier**: Private fields are accessible to all methods within the same class, regardless of which instance is being accessed.
   - **`this`**: `this` **cannot** be used in `static` method.
   - **`final`**:
     - In Java, a `final` field means that once it's assigned a value, it cannot be changed. However, you can (and must) initialize it either **at the point of declaration** or **in the constructor**. The key is that the assignment has to happen exactly once, and after that, the value is locked in.
     - `final` in a **field declaration** prevents **re-assignment**, in a **class declaration** prevents **inheritance**, in a **method declaration** prevents **overriding**.
   - **Modifier Order**: an example is `public static final void`, this is to declare a **constant**
   - **Print Modifier**:

Specifier	**Data Type**
$\%d$	Integer (decimal)
$\%f$	Floating point (decimal)
$\%s$	String
$\%c$	Character
$\%b$	Boolean
$\%\%$	Literal % sign

   To print the decimal point, we use `%.2f`(2 decimal places)
2. **Inheritance**:
   - The constructor of the subclass **should** invoke the constructor of the superclass via `super()`
   - **`super`**: Besides the use in the constructor, `super` should also be used when we want to call the method from the superclass. (According to information hiding, usually the fields of the superclass are not public)
   - Suppose we have two classes P and Q, if Q inherits from P, then we can say Q is the **subtype** of P or Q $<:$ P.
3. **Override vs. Overload**
   - **Override**: must have same **method descriptor (method signature + method return type)**
   - **Overload**: must have same **method name**, in the same class and **different method signature (method name, number of parameters, type of each parameter, order of the parameters)**
   - In the **subclass** of an **abstract class**, you still can **override the concrete method in that abstract class**.
4. **Abstract class**: An abstract class in Java is a class that has been made into something so general that it **cannot be instantiated**! And it **can** have the following:
   - **Abstract method**: An abstract method **should not have** any method body but it **may throw an exception**! An abstract class without an abstract method is also allowed!
   - **Concrete method**: As the name suggests, methods that are **not** abstract are concrete!
   - **Instance/Class Field**: fields with `static` or without.
5. **Concrete Class**:
   - a concrete class must have **implementations for all inherited abstract methods** (if it extends an abstract class).
   - Beyond that, it's free to have whatever you want — or even nothing at all in terms of fields or methods — since Java doesn't mandate that a class contain anything specific to be concrete.
6. **Interface**:
   - **Declaration**: The declaration of an interface should begin with keyword `interface`
   - **All** methods declared in an interface are `public abstract` by default. To declare an method in the interface, use e.g. `void foo();`
   - Interface **cannot** have **fields** and **concrete methods**!
7. **`Object::equals`**: It will compare whether two objects are referenced to the **same memory address** or not. **Note**: To override this function from `Object` so it behaves as we want, we need to
   - check the RTT of `obj` is a **subtype** of the type we are interested (can be generic type), by using `if (obj instanceOf TYPE)`, if the TYPE is a generic type, it **must be an unbounded generic type, e.g. `A<?>`, it cannot be `A<String>`**
   - typecast `obj` to the type we are interested by using either the class name or generic type with unbounded wildcard, e.g. `Box<?>`, **always be careful when when you want to type cast to a generic type, since you are casting it to a rawtype!**
8. **`Comparable<T>::compareTo(T t)`**: the return type of this method is `int`.
9. **OOP Design Tips**:
   - Identify the **nouns** (these tell what **classes** you need).
   - Set up the **relationship between the classes**. (**composition** or **inheritance** or unrelated)
   - Identify the **properties** and/or **data** needed to accurately describe the objects identified in Step 1.
   - Identify the functionalities and **bahaviour** of each class, i.e. what does this class do? (these tell you the **methods** for each class)
   - **Single Responsibility Principle**: Each class should only be responsible for doing one single thing.
   - **Consecutive Unique ID**: This can be done by `private static int next = 0`, `private final int id` Then inside the constructor, use `this.id = next`, `next += 1`
   - **The elegant use of `toString()`**: if your class has a `String` field that you want to get from outside, you can encapsulate it into `toString()` method of the class, so that calling the class itself by using either `this` or `super` will give you that string.

Exception & Wrapper Class

1. **Application of CTT and RTT**
   - (**CTT**): To see whether a code will generate compile-error or not, we only see the CTT of the variable and the **type casting**.
   - (**RTT**): **Run-time** error judgment **only** needs us to see the **RTT** of the variable. We **must ignore** the type casting because Java is **strongly typed**, meaning objects always retain their actual type (RTT).
2. **Exception**
   - **Unchecked Exception**: It is a subclass of `RuntimeException`, which is a subclass of `Exception`. **Not necessary to be handled** but it is recommended to do so.
   - **Checked Exception**: It is a subclass of `Exception`. **Must be handled**.
   - **Throw an exception**: Use the syntax `throw new specificexception();`
   - **Define a method that may throw an exception**: Whenever a method may throw an exception, use **`throws`** `specificexception` after the parameters. e.g. `public void move(double distance) throws CannotMoveException`
   - **Handle the exception**: This **must be done** in the **`catch`** block or be passed to another "catch" block. If there is no need for the `finally` block, can omit it.
   - **Pass messages to be shown to `Exception`**: e.g. `super(String.format("Cannot set volume to %d", volume));`, where `volume` is a parameter.
   - **FileNotFoundException**: Use `import java.io.FileNotFoundException`
   - **Get the Exception's Message**: In `Exception` and its subclasses (denote the specific exception as `e`), there is a `String` field called `Message` and to get the `String`, we can use `e.getMessage()`

Generics & Wildcards

1. **Generic Type**:
   - **Constructor**: the constructor of a generic type shouldn't contain <> operator. **Note**: when we **call** the constructor, we **must include <> operator**
   - **Factory method**: it is a **class** method (declared with `static`) and a **generic** method (declare a method-level type parameter). e.g., `public static <T> Box<T> of(T obj) { return new Box<T>(obj); }`
   - **Parameterize a generic type**:
     - When we use `extends` or `implements` a generic type, we **must** instantiate the generic type!
     - When we call a method from a generic type, we should also **parameterize** the generic type either explicitly, e.g. `Box<String>`, or implicitly, e.g. `Box<>` (**must include <>**)
     - **Rule of Thumb**: Always think about **which generic type is the one you want to instantiate**!
   - **Subtype between generic type**: If you explicitly use `extends/implements`, e.g. `class A<T> extends B<T>`, then `A<T>` is a **subtype** of `B<T>`.
   - **Use `Object` to ensure the generalizability, if generic types are too tedious**.
2. **Generic method**:
   - **Non-static Generic method**: e.g. `public <U> Box<U> map {}`, `public Box<S> map {}`, this kind of method **may or may not** declare **method-level** type parameter, it can use **class-level** type parameter. And it depends on design requirements.
     - **Invoke**: To invoke, we can use `instance.method()`
   - **Static Generic method**: e.g. `public static <T> Box<T> ofNullable(T obj) { }`, this kind of method **must be declared using a method-level type parameter**.
     - **Invoke**: To invoke, we can use `ClassName.<Type>method()`, or we can **omit** the `<Type>` to let the compiler do the type inference.
   - **Field-leve type parameter**: Java **doesn't have** field-level type parameter!
3. **Generic Array**:
   - we **cannot instantiate** a Java array using the type parameter, e.g. `new T[]` is not allowed. However, we **can declare** a Java array using the type parameter, e.g. `T[] a` is allowed.
   - **An example**: `@SuppressWarnings("unchecked")`, then `Queue<Passenger>[] temp = (Queue<Passenger>[]) new Queue<?>[totalStops]`

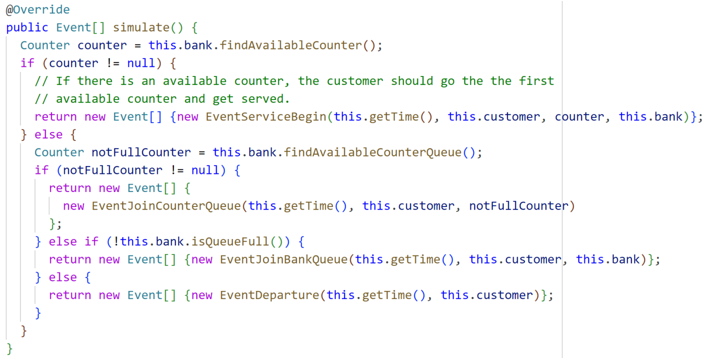
- The generic array you declared after using the above method is **nothing but a Java Array**, it has `length` property!

4. **PECS Rule**: Producer `extends`, consumer `super`. **Note that PECS is usually used on method parameter.** An easy way to think of it is as follows
   - Take the method parameter as your `studyObject`
   - look at the `studyObject.method()`
   - If `.method()` is something like `get()`, `read()`, then your `studyObject` is a producer, add **lower-bounded wildcard** to your method parameter.
   - If `.method()` is something like `set()`, `write()`, then your `studyObject` is a consumer, add **upper-bounded wildcard** to your method parameter.
5. **Wildcards**
   - Wildcards **is not a type!**, so, you **cannot** use them in class declaration and **cannot use them as type arguments!**! But wildcards **can be used to instantiate an array of generic types**.
   - **The following is not allowed**
     - `private static final Box<?> emptyBox = new Box<?>(null);`
     - `public <T> of(<? extends T>[], int depth)`
   - **Unbounded Wildcards**: Always use <?> instead of raw types when you need to check generic types with `instanceof` or instantiate an array of generic types.

Classic PE Questions

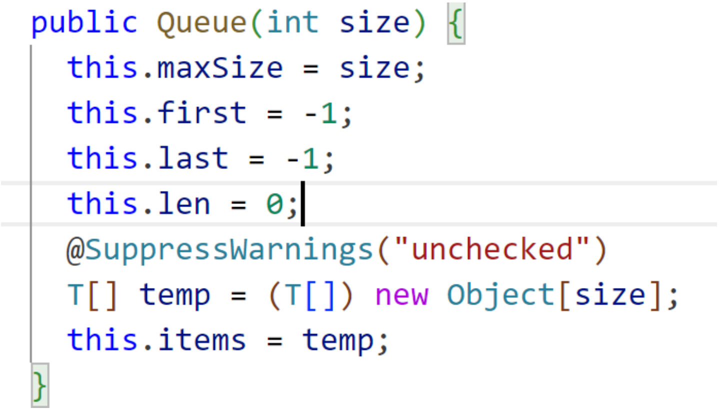
1. **Simulation**:
   - In Simulation, we don't have to care about the sequence of the events in simulation thanks to the use of priority queue, which used the time as the key! So, we just have to **think about how one event will trigger the others and what fields should an event has**!
   - **Always think about what fields should an event have and how an event will transit to another!**

Event Design



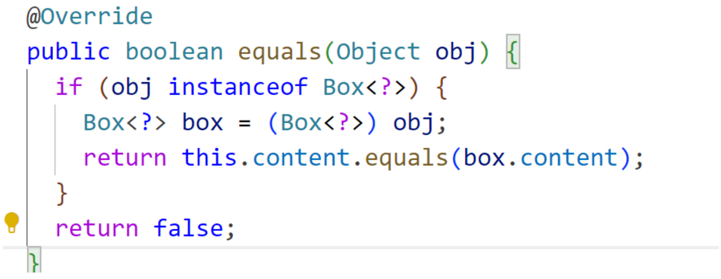
```java
@Override
public Event[] simulate() {
  Counter counter = this.bank.findAvailableCounter();
  if (counter != null) {
    // If there is an available counter, the customer should go the the first
    // available counter and get served.
    return new Event[] {new EventServiceBegin(this.getTime(), this.customer, counter, this.bank)};
  } else {
    Counter notFullCounter = this.bank.findAvailableCounterQueue();
    if (notFullCounter != null) {
      return new Event[] {
        new EventJoinCounterQueue(this.getTime(), this.customer, notFullCounter)
      };
    } else if (!this.bank.isQueueFull()) {
      return new Event[] {new EventJoinBankQueue(this.getTime(), this.customer, this.bank)};
    } else {
      return new Event[] {new EventDeparture(this.getTime(), this.customer)};
    }
  }
}
```

Queue



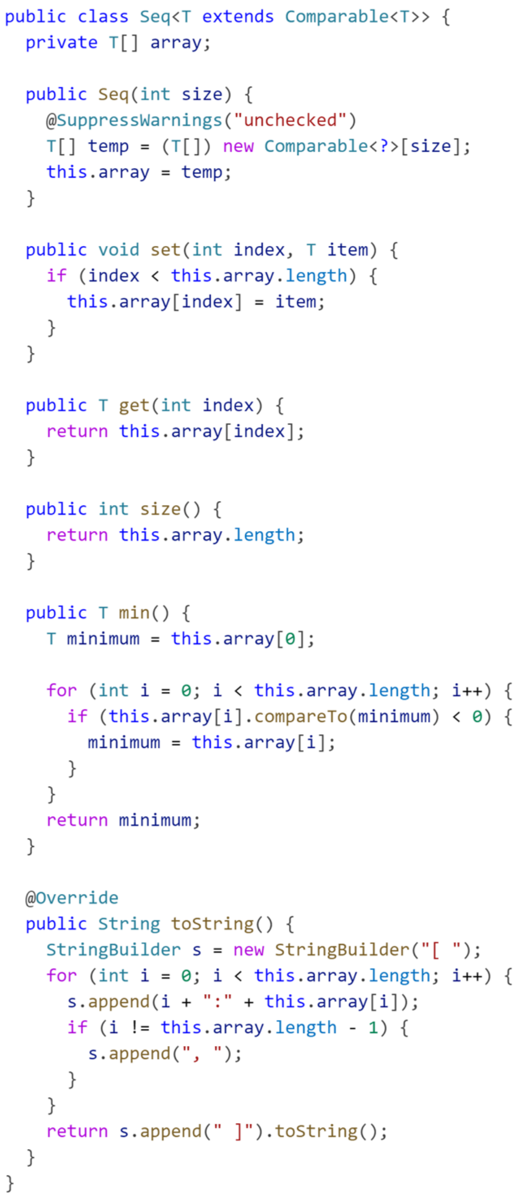
```java
public Queue(int size) {
  this.maxSize = size;
  this.first = -1;
  this.last = -1;
  this.len = 0;
  @SuppressWarnings("unchecked")
  T[] temp = (T[]) new Object[size];
  this.items = temp;
}
```

equals design



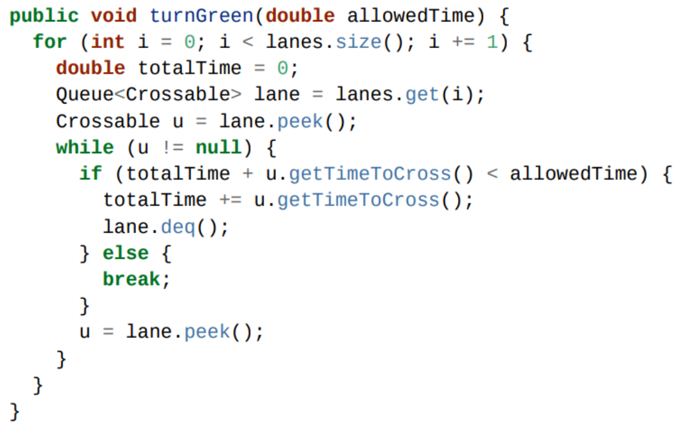
```java
@Override
public boolean equals(Object obj) {
  if (obj instanceof Box<?>) {
    Box<?> box = (Box<?>) obj;
    return this.content.equals(box.content);
  }
  return false;
}
```

Seq



```java
public class Seq<T extends Comparable<T>> {
  private T[] array;

  public Seq(int size) {
    @SuppressWarnings("unchecked")
    T[] temp = (T[]) new Comparable<?>[size];
    this.array = temp;
  }

  public void set(int index, T item) {
    if (index < this.array.length) {
      this.array[index] = item;
    }
  }

  public T get(int index) {
    return this.array[index];
  }

  public int size() {
    return this.array.length;
  }

  public T min() {
    T minimum = this.array[0];

    for (int i = 0; i < this.array.length; i++) {
      if (this.array[i].compareTo(minimum) < 0) {
        minimum = this.array[i];
      }
    }
    return minimum;
  }

  @Override
  public String toString() {
    StringBuilder s = new StringBuilder("[ ");
    for (int i = 0; i < this.array.length; i++) {
      s.append(i + ":" + this.array[i]);
      if (i != this.array.length - 1) {
        s.append(", ");
      }
    }
    return s.append(" ]").toString();
  }
}
```

Iterate through the Queue



```java
public void turnGreen(double allowedTime) {
  for (int i = 0; i < lanes.size(); i += 1) {
    double totalTime = 0;
    Queue<Crossable> lane = lanes.get(i);
    Crossable u = lane.peek();
    while (u != null) {
      if (totalTime + u.getTimeToCross() < allowedTime) {
        totalTime += u.getTimeToCross();
        lane.deq();
      } else {
        break;
      }
      u = lane.peek();
    }
  }
}
```